# Module 4: Trusted Execution Environments

## Task 2: Secure Communication Channel [20 Points]

---

Prof. Shweta Shinde *shweta.shivajishinde@inf.ethz.ch*
Supraja Sridhara *supraja.sridhara@inf.ethz.ch*
Mark Kuhne *mark.kuhne@inf.ethz.ch*

Welcome to the fourth module of the Information Security Lab. This task is part of the Module-04 on *Trusted Execution Environments*, or TEEs in short.

## 1 Overview

TEEs, for example Intel SGX, isolate user-space programs in secure *enclaves* that do not trust the operating system. TEEs protect program execution inside an enclave by assuming hardware support for several operations, including memory isolation. The hardware-based attestation can be used to setup shared secrets between entities to build secure channels.

### 1.1 Task Summary

This task explores the importance of secure communication channels between multiple enclaves. For this task, consider an Enclave E that performs secure computation and stores results in a remote secure storage peripheral P i.e., E and P do not run on the same machine. E issues store/retrieve commands to P over a channel C.

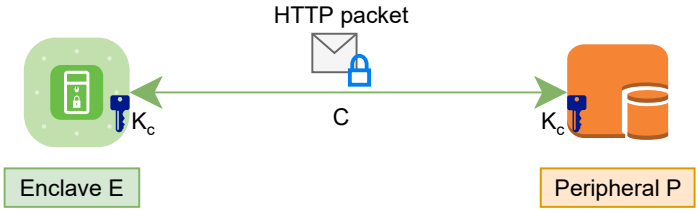


Figure 1: The Enclave E and Peripheral P communicate over the network by exchanging HTTP packets over the channel C. E and P use a pre-established key $K_c$ to encrypt and integrity protect the messages which are encapsulated in the HTTP packet.

**Enforcing Isolation.** For this task, we don't have access to a hardware based TEE. Instead, we have emulated the isolation by making the components (E and P) black-box and non-debuggable. In other words, you do not have `sudo` access and thus cannot perform any operations that a compromised OS could. So, for all practical purposes for this task, these user processes are

considered to be secure enclaves. Note that, for simplicity, we run the E and P processes on the same machine.

**Attacker Capabilities.** Under the above setting, you (the attacker) cannot do much to E or P. However, *you have all the capabilities of a network attacker under the Dolev–Yao model*.

**Goal.** The goal of this task is to study the properties of a secure channel under the attacker model described above. Specifically, you will be performing the following steps:

(a) Investigate if there are any protocol vulnerabilities in the communication.

(b) Investigate if there are any implementation vulnerabilities in the channel.

(c) Implement real exploits to demonstrate the feasibility of end-to-end attacks identified in (a) and (b).

# 2 Getting Started

For this task you should download VM_Task2.ova from Moodle and create a new virtual machine (VM) in VirtualBox. For more details on how to download and install VirtualBox and create a new VM from the .ova refer to the exercise session handout on Moodle. We strongly recommend that you use the default configuration (RAM, CPU and network) in the .ova file. This is important so as to ensure that the timing measurements you do locally (e.g., total execution time) will be the same on our grading VM. We will use the same VM configurations to grade your solutions.

## 2.1 Testing your setup

Once you have created the VM from VM_Task2.ova and started it in VirtualBox, you can boot the VM. To test your setup follow these steps:

1. Execute `cd /home/isl && tree -L 2`. You should see the folder structure shown in Listing 1. Each directory and file in this tree has a specific purpose, as explained below.
    - `user`: your ETH id e.g., `11-111-111` is written into this file. The value from this file is read by different binaries to generate user-specific flags.
    - `scripts`: this folder is empty. The grading script runs your solution script from this folder. See Section 4.4 for more details on the grading scripts.
    - `t2`: this folder contains all binaries and scripts required for this task.
        - `enclave`, `peripheral`: Executable binaries for E and P respectively. These are binaries i.e. you do not have access to the source code. Please do not modify or replace them while solving the task. Your solution script might not pass in our grader if you use replaced/modified binaries, see Section 4.3 for details.

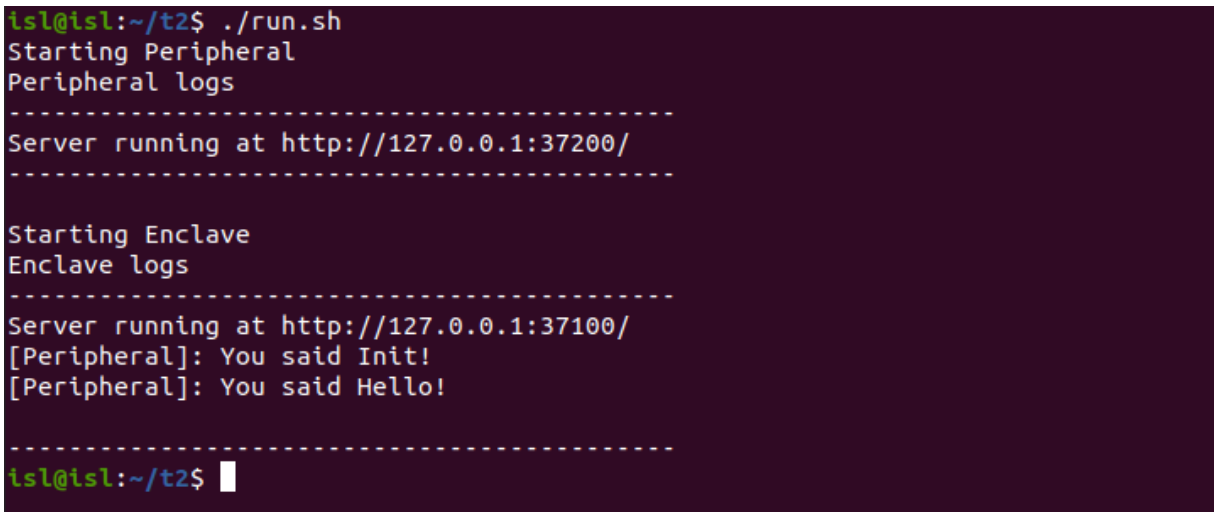


Figure 2: Example execution of `/home/isl/t2/run.sh`

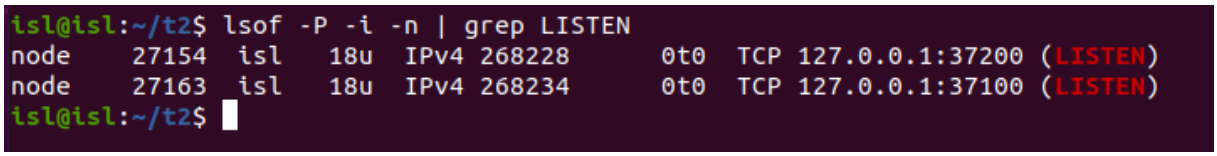


Figure 3: E and P listen on ports 37100 and 37200 respectively

- `test_setup.py`, `test_enclave.py`, `test_peripheral.py`: these python scripts are used to test your setup and debug it in case of issues. Read Section 2.2 for more details on these scripts.
- `run.sh`: Kills any running E or P processes and starts them again. This script also creats log files `/home/isl/t2/enclave.log` and `/home/isl/t2/peripheral.log` for E and P respectively.

2. Execute `./t2/run.sh`. This script starts E and P. If the processes are started correctly, the script displays a success message and lists the open ports for all the components as shown in Figure 2. If the script fails please read Section 2.2 to debug your setup.

3. Execute `lsof -P -i -n| grep LISTEN`. You should see E and P as node processes listening on the ports as shown in Figure 3.

Listing 1: Directory structure of VM_Task2.ova

```
1 isl/
2 ├──user/
3 ├──scripts/
4 └── t2/
5     ├── enclave
6     ├── peripheral
7     ├── test_setup.py
8     ├── test_enclave.py
9     ├── test_peripheral.py
```

### 2.2 Debugging your setup

1. If executing `/home/isl/t2/run.sh` fails, ensure that the ports needed by the application are not held by any other processes. To check processes listening on a port execute `lsof -i:<port_number>`. We have ensured that E and P use distinct port numbers. If you have installed or written programs that use these ports, please modify them to use other ports. To kill processes using the port execute `kill -9 $(lsof -t -i:<port_number>)`. Replace `<port_number>` with the port number you are interested in.

2. On executing `/home/isl/t2/run.sh`, if you do not see any messages being exchanged between E and P in their log files, check that the components are running correctly. For this, you can run the `test_setup.py` script which tests all the components and reports their status. To investigate the components individually, run the corresponding test script in `/home/isl/t2` folder.

   Individual test scripts `test_*.py` send a healthcheck messages to the ports that the respective components listen on. The components respond with success or failure messages. The response messages are printed to the console and should help you identify the problem with the components.

3. If any of the components hang or crash because of your experimental executions, run the `/home/isl/t2/run.sh` script and follow the steps in Section 2.1.

4. Please note that the programs do not have any persistent state. So, if you restart them, your execution state will not be stored or restored. You will have to redo any actions you had performed before the restart.

## 3 Task Description

### 3.1 Setup

As depicted in Figure 1, the Enclave E and secure storage Peripheral P communicate over a HTTP channel C. E and P use AES in CBC mode to encrypt the messages using a pre-established secret $K_c$ and add SHA-256 MAC tags before sending them over this channel. These protected messages are encapsulated in the HTTP packets, see Figure 4, sent over C. E and P run in secure enclaves and their execution cannot be compromised by an adversary defined in Section 1.1. For this task, we assume that the peripheral P only stores data for a single enclave E.

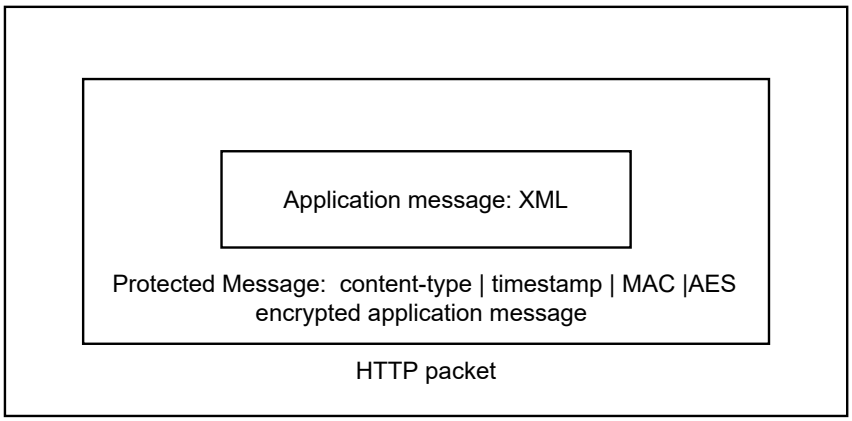


Figure 4: The Application message in XML format is encrypted. It is added to the Protected Message along with content-type, timestamp, and MAC. The Protected Message is then sent in the body of the HTTP POST packet.

## 3.2 Communication

The communication between E and P proceeds as shown in Figure 5. The messages and responses are exchanged via protected messages as shown in Figure 6. Each of the message and response types have a specific message format. These are described in Section 3.3.

**Message-Response types and Admin commands.**

- Message types: `hello,store,gets`.
- Response type: `hi,done,puts`.
- Admin commands: can be any string (e.g., `init` and `stop`).

## 3.3 Message format

Listings 2–5 show the message formats for the messages shown in Figure 5. When an attribute has value "`*`", it can be any string and is not constrained. However, the message length cannot be more than $352$ bytes. If the attribute values are listed in curly braces, for example `{'str1', 'str2', 'str3'}`, then the value can be only one of the strings—`'str1'` or `'str2'` or `'str3'`. Note that the white-spaces in the Listings below are to improve readability. E and P send the XMLs are without white-spaces.

Listing 2: Message type : hello

```
1 <start_messages>
2         <mes cd="hello"></mes>
3 </start_messages>
```

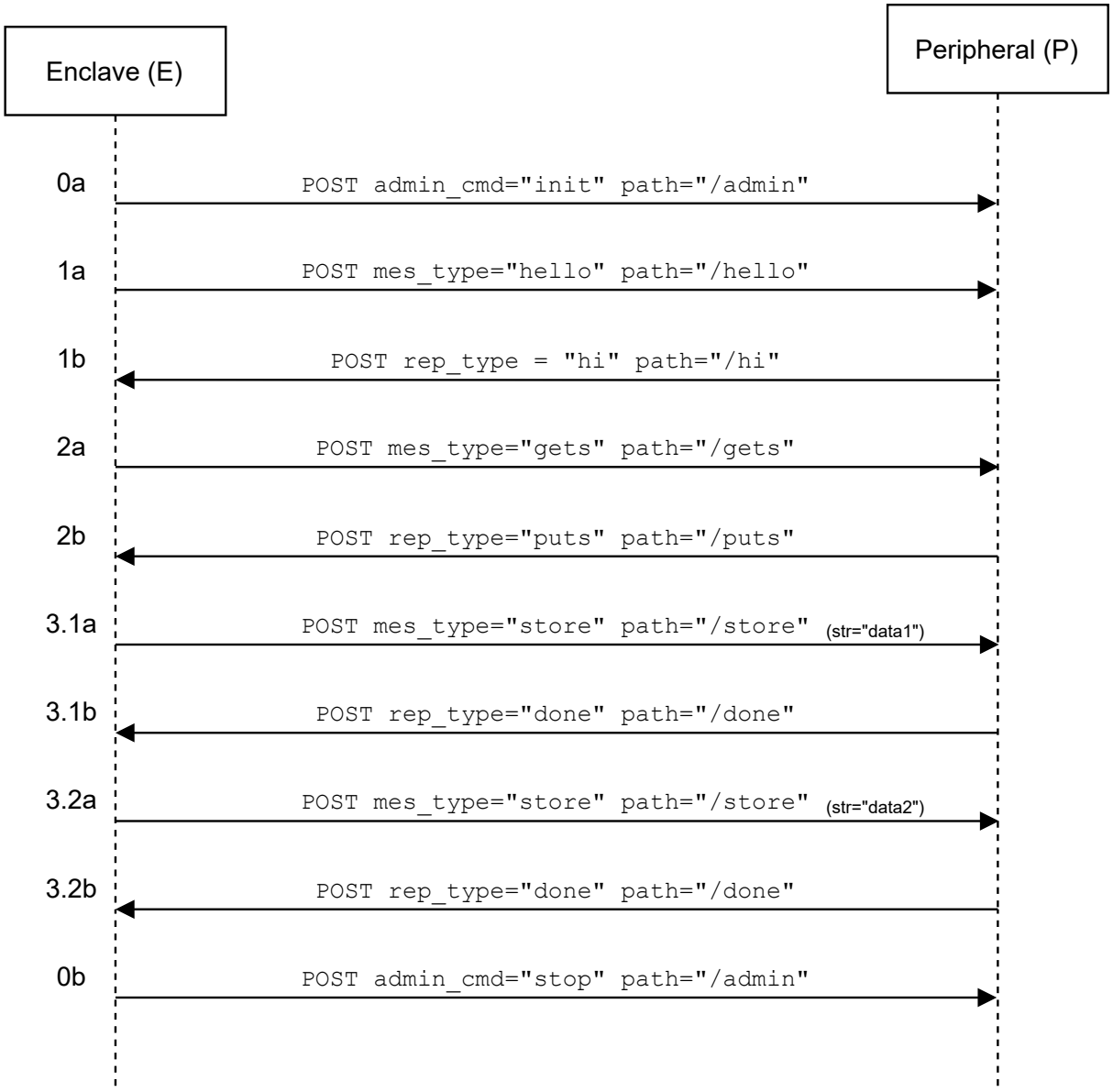


Figure 5: Communication between E and P proceeds as message response pairs. E initiates the communication by sending a message of type `hello` to P and waits for a response of type `hi` from P. It then sends a `gets` message to retrieve data it had stored in its previous execution, to which P responds with a `puts` message. Further, E sends the `store` message with string `"data1"` and waits for the `done` message from P. On receiving `done`, E sends another `store` message with string `"data2"` to which P responds with `done`. Additionally, E can use admin commands `init` and `stop` to initialize and stop the communication with P respectively.

Listing 3: Response type : hi

```
<start_messages>
        <mes cd="hi"></mes>
</start_messages>
```

Listing 4: Message type : gets

```
<start_messages>
        <mes cd="gets"></mes>
</start_messages>
```

Listing 5: Response type : puts

```
<start_messages>
        <mes cd="puts" str="*"></mes>
</start_messages>
```

Listing 6: Message type : store

```
<start_messages>
        <mes cd="store_str" str="*"></mes>
</start_messages>
```

Listing 7: Response type : done

```
<start_messages>
        <mes cd="done" str="*"></mes>
</start_messages>
```

In addition to the above message and response types, E can send admin commands to P. This format is shown in Listing 8. The `cd` attribute of the `mes` element is always a string of $4$ ASCII characters.

Listing 8: Admin command

```
<start_cmd>
        <mes cd="*"></mes>
</start_cmd>
```

### 3.4 Protected message format

The messages described in Section 3.3 are encrypted using AES in CBC mode with a hex-encoded constant IV, as shown in Listing 11.The encryption results in a $256$ bytes long hex-encoded string that forms the message body, as depicted in Figure 6. A MAC tag is computed

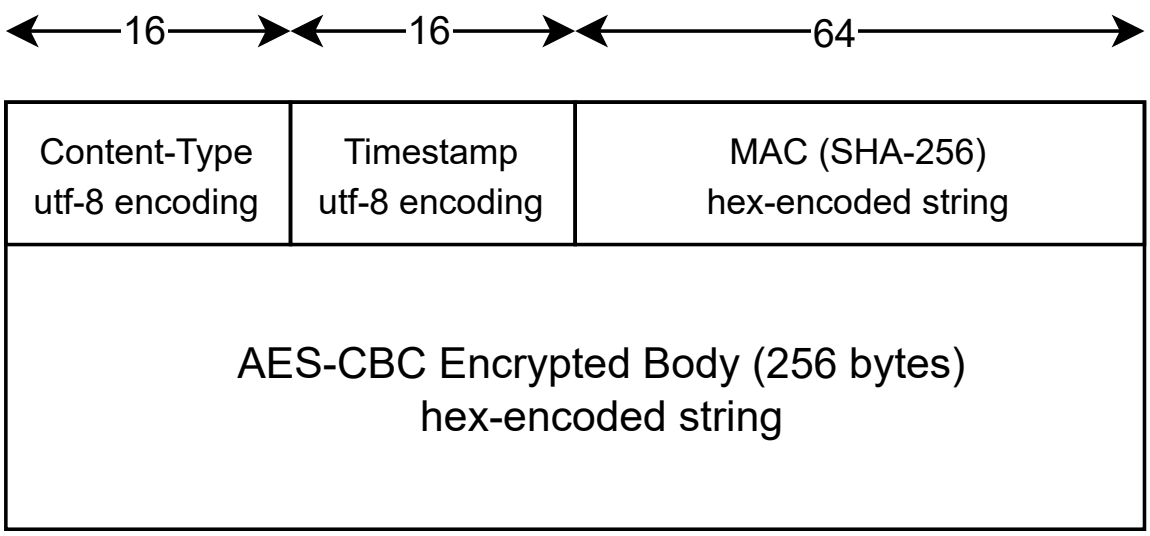


Figure 6: Protected message format. The messages from Section 3.3 are encrypted using AES-CBC mode and added as $256$ bytes body to the protected message. A MAC tag is computed over the body and added to the header of the protected message along with the content type and timestamp.

over the $256$ bytes of message body using SHA-256 resulting in a $64$ byte long hex-encoded string that is added to the header of the protected message. The header also contains a $16$ byte long utf-8 string for Content-Type and a $16$ byte long utf-8 string for timestamp. This protected message is depicted in Figure 6. The fields of the protected message are explained in detail below.

- Content-Type: A 16 byte utf-8 type string that the protected message contains. The value of this field is a message or response type string from Section 3.2. If the protected message contains an admin command, the value of this field is `admin`. The type string is terminated with a dollar symbol $\$$ and then padded to $16$ bytes using ASCII-0s e.g., if the type is `admin` then the string would be `admin$0000000000`. This padded string is added as the first $16$ bytes of the protected message.
- Timestamp: A Unix timestamp of when the protected message is created. The Unix timestamp is converted to a utf-8 encoded string, terminated using a dollar symbol $, and then padded to $16$ bytes using ASCII-0s ('0'). This is the same process as explained above for padding the content-type. The timestamp is added as the second $16$ bytes field to the protected message.
- AES-CBC Encrypted Body: XML message is padded using $0\text{x}00$, and not a ASCII-0s, to get a $224$ byte hex-encoded message. This is then encrypted using AES in CBC mode with the key $K_c$. Note that the first $32$ bytes of the $256$ bytes message body are for the IV i.e., the message can be a maximum of $224$ hex-encoded characters long.
- MAC: A MAC tag created over the $256$ bytes using SHA-256 resulting in a $64$ byte long hex-encoded tag string.

Next, we show the pseudo code in a Python-like syntax to demonstrate how the protected messages are generated. Please note that these listings are only provided for clarity. While

forming your protected messages, it is your responsibility to ensure that the protected message adheres to the format shown in Figure 6.

Listing 9: Content-Type

```
mes_type=['hello','store','gets']
rep_type=['hi','done','puts']
#|| signifies logical OR
type_str = mes_type[*] || rep_type[*] || "admin"
#the type_str is padded to 16 bytes by $ and 0s
content-type = bytes(type_str + '$' + '0' * (16-len(type)-1), 'utf-8')
```

Listing 10: Timestamp

```
timestamp_str = str(int(time.time()))
timestamp = bytes(timestamp_str + '$' + '0' * (16-len(timestamp_str)-1), '↩
    utf-8')
```

Listing 11: Encrypted body

```
pad(message) {
    #pad hex encoded message to 224 bytes with 0x00s
    padded_message[224] = message + ((224-len(message)) * 0x00)
}
#the message is padded and then encrypted using AES in CBC mode with key ↩
    Kc and converted to bytes
iv = 11111111111111111111111111111111
#enc-body is a 256 bytes long hex-encoded string
enc-body = iv + (AES-CBC_{Kc}(pad(message), IV=iv)).to_bytes(224,'big')
```

Listing 12: MAC

```
#mac is a 64 bytes long hex-encoded string
mac = (SHA-256(enc-body)).to_bytes(64,'big')
```

Listing 13: Protected Message

```
pkt = content-type + timestamp + mac + enc-body
```

### 3.5 Approach

**Start communication.** To trigger the communication between E and P, as shown in Figure 5, execute `./home/isl/t2/run.sh`. This script starts the E and P processes. Once E starts

executing, it starts the communication with P by sending an `init` message. This communication is shown in Figure 5 step 0a. When E and P execute, they immediately exchange the entire sequence of messages (0a-3b in Figure 5). To stop the communication, E sends the `stop` message to P (0b in Figure 5). At this point, P clears all its local state except for the strings stored in it. Note that, our implementation of E executes the communication sequence between E and P only once. However, we do not terminate E and P, they continue their execution to listen for messages on their respective ports.

**Burp Suite.** We have installed Burp Suite on the task VM. This will allow you to intercept and analyze the messages being exchanged between E and P. For more details on how to use Burp Suite refer to the exercise session handout on Moodle.

**Extracting flags.** For this task you should think like an attacker (c.f. Section 1.1) who is constrained under the Dolev–Yao model for channel C. We have planted four flags in the peripheral P for this task. It's your job to identify possible attacks. Specifically, you have to implement these attacks in order to trick P into creating the flag files.

On a successful attack, the peripheral writes a flag file `flag-2-[1..4]` to `/home/isl/scripts`. Note that, you do not have to explicitly extract the flag files from P. Instead, your attacks should trick P into automatically creating the flag files. See Section 4.2 for more details on the generated flag files.

# 4 Grading

## 4.1 Submission format

For this task you should implement all your end-to-end exploits in a single python file named `submit-2.py`. Please write your code using Python v3.8.10 that is pre-installed in the task VM. You should submit this python file on Moodle.

## 4.2 Grader program

To evaluate your solution, our grader program will run `submit-2.py` and read the flag files that the peripheral creates.

Specifically, our grader program downloads a fresh copy of the VM image that was provided to you and boots it. Next, it copies your script `submit-2.py` to `/home/isl/scripts` in the fresh VM it booted. It then executes `python3 submit-2.py >& /dev/null` on the VM and copies out the flag files (flag-2-[1..4]) from the `/home/isl/scripts` folder to our machine. The files are checked and evaluated to calculate the final points for the task. You can run this grader program on your system by downloading the grader-2.zip from moodle. The grader program is distributed as a .zip file with the structure shown in Listing 14.

Listing 14: Structure of grader-2.zip

```
grader-2.zip
├──grade.sh
├──check_flags.py
└──README
```

**Running the grader program yourself.** Do not perform the following steps to run the grader program inside the task VM. You will have to use a Linux host machine with Ubuntu 20.04.3 LTS to run the grader program with VirtualBox installed.

To run the grader, extract the `grader-2.zip` to `grader-2` folder on your Linux host machine. Next, follow the steps below:

- Download a fresh copy of the task VM (`VM_Task2.ova`) and copy it to `<path to grader-2>`. This is so that the grader program does not need to download the VM every time.
- execute `chmod +x <path to grader-2>/grade.sh` provide execute permissions for the script.
- execute `<path to grader-2>/grade.sh <path to submit-2.py>`. The `grade.sh` script reads the path to the solution python script as the first command line argument.
- execute `<path to grader-2>/check_flags.py <eth-id>` that checks the flag files and grades them. The final grade file `grade-<eth-id>` is created in the same directory with points for each flag. See Figure 7 for an example execution of the grader program. Note that, this output is truncated.
- Note that if you run the grader program multiple times, it will end up using the same VM. If you want to test it on a fresh VM, delete the grading VM named `VM_Task2_Grader` that was created by the grader program.

### 4.3 Writing your exploits

1. For this task, given the attacker capabilities in Section 1.1, the binaries for `E` and `P` are treated as black boxes that cannot be reversed. Trying to do this will not give you any useful results and is not a part of the task.
2. The flag files `flag-2-*` created by the peripheral `P` depend on your ETH student ID, for example, 11-111-111. You can put your student ID in the file `/home/isl/user`. However, please note that the example grader works for the example student id 11-111-111. So, if you use your own ID, the grader will fail—this is an expected outcome.
3. The peripheral we use for final grading uses a different approach to generate the flags based on individual student IDs. So hard coding the flags you get in your script or trying to reverse-engineer the flag generation will fail when we do the final grading.

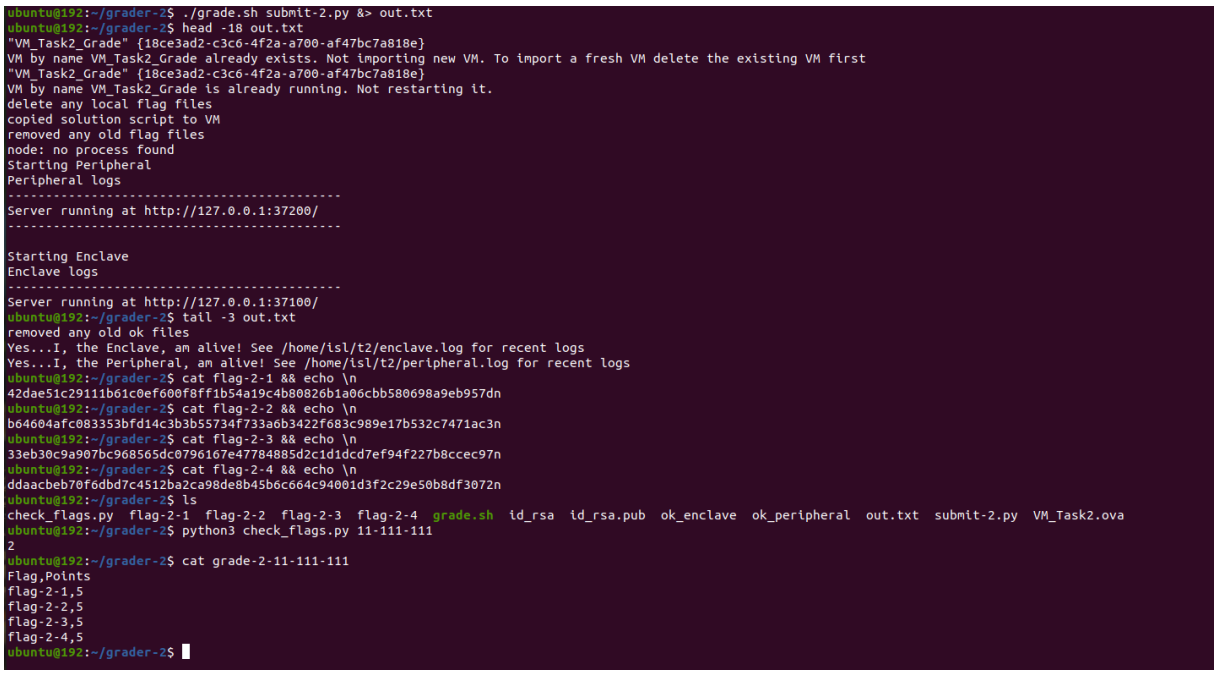


Figure 7: Example execution of the grader program

4. The script you submit, `submit-2.py`, should not modify the flag files created by P.
5. The script you submit, `submit-2.py`, is responsible for starting the E and P processes. Our grader program will first check that these processes are running and then check the flag files.
6. The grader program terminates all scripts after 30 seconds.

### 4.4 [20 Points] Grading criteria

For each successful flag, our auto-grader will give you 5 points. So, you can get a maximum of 20 points for this task.

We will only execute your submission file for a maximum of 30 seconds. After 30 seconds, our grader program will kill your script and then read the `flag-2-[1..4]` files created by P to calculate the points for this task. If the `flag` files were not created by P, the grader program simply ignores the missing files. The grader program computes the points for the `flag-2-*` files it found in the `/home/isl/scripts` folder.

Please note that we will use a clean environment for each script. Our grading VM will have the exact same environment as the VM that we gave you. The python script you submit should be self contained. We will not setup special execution environments (e.g., path variables) or debug your scripts. We will spawn a fresh VM to grade each submission. Do not assume that our grading machine will have:

1. any libraries or environment variables that are not available on the VM.
2. sudo or root permissions for running your script.
3. internet to connect to other machines.

4. open ports to launch and execute other scripts, daemons, or services.

5. persistent storage on our machine.

The flag files (`flag-2-[1..4]`) are auto-generated. Do not modify them. Any intentional attempts to write to these files will be detected and the grader program will terminate without evaluating your solution. Furthermore, any attempts to corrupt the machine our grader VM or host will result in zero points for this module. In addition, please carefully read Section 4.3 above.

# 5 Discussions

Friess Carl (carl.friess@inf.ethz.ch), Suter-Dörig Andris (sandris@student.ethz.ch), Mink Eric (minker@student.ethz.ch), Mark Kuhne (mark.kuhne@student.ethz.ch), Supraja Sridhara (supraja.sridhara@inf.ethz.ch) and Shweta Shinde (shweta.shivajishinde@inf.ethz.ch) will monitor the Moodle discussions. We encourage you to ask clarification questions on this public forum. That way everyone will benefit from clarifications we provide and it reduces duplicate questions. If you have a question that requires explaining your solution, please send a private email to all of us. Note that we will only answer questions that genuinely warrant a response. As a rule of thumb, we will not provide hints or help you to debug your code.
In conclusion, happy cracking & coding!